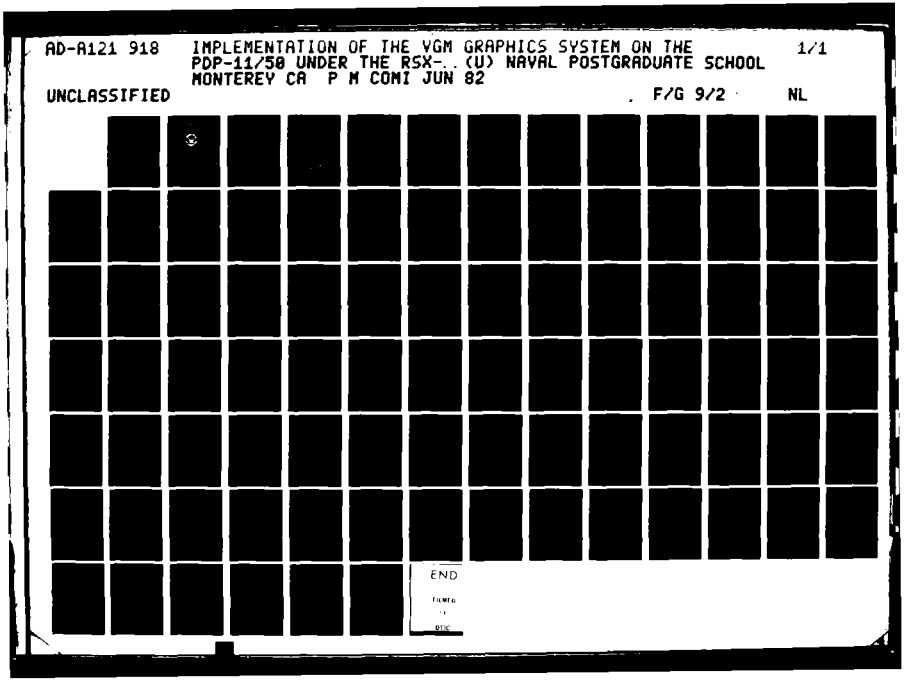
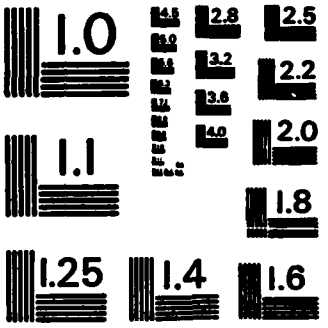
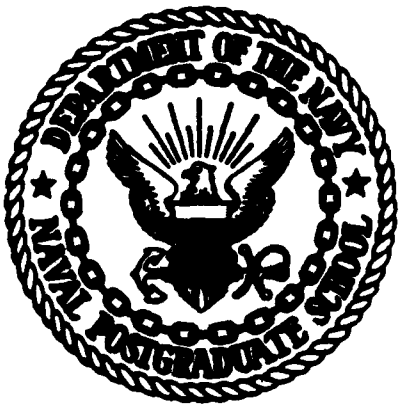

AD-A121 918
IMPLEMENTATION OF THE VGM GRAPHICS SYSTEM ON THE PDP-11/50 UNDER THE RSX-.. (U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P M COMI JUN 82
1/1
UNCLASSIFIED
F/G 9/2
NL
END
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

Implementation of the VGM Graphics System on the PDP-11/50 Under the RSX-11M Operating System and Construction of a Compatible Software Driver for the Ramtek RM-9400

by

Patrick Michael Comi

June, 1982

Thesis Advisor: G. A. Rahe

Approved for public release, distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A121918	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Implementation of the VGM Graphics System on the PDP-11/50 Under the RSX-11M Operating System and Construction of a Compatible Software Driver for the Ramtex RM-9400	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June, 1982
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Patrick Michael Comi	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	12. REPORT DATE June, 1982
	13. NUMBER OF PAGES 87
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer graphics, CORE graphics system, graphics standardization, graphics program portability, Virtual Graphics Machine

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In 1977 the ACM Special Interest Group for Graphics (SIGGRAPH) formed the Graphics Standards Planning Committee (GSPC) to develop a standard for the industry. The result of their efforts was the CORE graphics system. This study discusses that system and the issues involved in its creation. It describes Bell Northern Research's approach to implementing CORE with their Virtual Graphics Machine (VGM).
The installation of VGM at the Naval Postgraduate School on a PDP 11/50 with the RSX-11M operating system is described as well as the initial efforts to expand it to drive the Ramtex RM-9400 Graphics

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED

Implementation of the VGM Graphics System on the PDP 11/50 Under the RSX-11M Operating System and Construction of a Compatible Software Driver for the Ramtek RM-9400

by

Patrick M. Comi
Lieutenant, United States Navy
B.S., Union College, 1970

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1982

Author: Patrick M. Comi

Approved by: _____
Thesis Advisor

_____
Second Reader

_____
Chairman, Department of Computer Science

_____
Dean of Information and Policy Sciences

ABSTRACT

In 1977 the ACM Special Interest Group for Graphics (SIGGRAPH) formed the Graphics Standards Planning Committee (GSPC) to develop a standard for the industry. The result of their efforts was the CORE graphics system. This study discusses that system and the issues involved in its creation. It describes Bell Northern Research's approach to implementing CORE with their Virtual Graphics Machine (VGM).

The installation of VGM at the Naval Postgraduate School on a PDP 11/50 with the RSX-11M operating system is described as well as the initial efforts to expand it to drive the Ramtek RM-9400 Graphics Display System.

TABLE OF CONTENTS

I. INTRODUCTION

Computer graphics is a relatively young technology and is expanding at an extremely rapid rate. As a recognized discipline, it marks its birth with Ivan E. Sutherland's SKETCHPAD, in 1962. As the field has expanded, hardware has developed along several different lines. The available devices range from computer driven mechanical pen and ink and photographic film plotters, to refresh display devices where digital stored images are repeatedly painted on a television-like Cathode Ray Tube (CRT). There are also storage tube displays where the CRT itself retains the image, thereby eliminating the need for storage and continuous refreshing. The latest development has been the raster-scan devices where a matrix of intensity values is output to a refresh type CRT.

The software supporting these various devices has developed along lines as diverse as the hardware. Despite wide variation in device capabilities there is a large body of graphics methodology that is common to all display systems. The existence of this body of shared technology has contributed significantly to the movement toward the development of a software system that would be common throughout the community of graphics programmers.

The concept of a standardized graphics system is not a new one. The publication of the ACM-SIGGRAPH Graphics Standards Planning Committee (GSPC) report in 1977, however, was the first widely accepted effort in the area of standardization. This CORE graphics system is, as yet, only a proposal. CORE is envisioned by the GSPC to be a first step toward a true, industry-wide standard. The hope for the current CORE system is that it will be implemented at a large number of computer graphics installations and be subjected to a variety of applications. Such widespread use of CORE is the best way to fully challenge the proposed standard. Extensive use of the system will build a comprehensive body of knowledge about it and should highlight its strengths and weaknesses. Based on such experience the system can be revised and restructured as necessary until ultimately it is accepted as a viable industry-wide standard.

This project is intended to be an initial step toward a thorough study of the CORE graphics system at the Naval Postgraduate School. The long-term goal of the research is to implement the GSPC proposed system on the graphics facility at the Naval Postgraduate School and critically examine its performance.

When this research was begun, exposure to CORE at the Naval Postgraduate School was limited to the information available in the literature. No version of it was

operational at the facility. It was expected that the progress of the work would be slow while experience with CORE was being accumulated. Since this study was the initial step, the emphasis was placed on producing a good foundation for work that would follow.

Naturally, the first step in the study of CORE was to implement a version of it. The PDP-11/50 computer and RSX-11M operating system were the target environment for this phase. The CORE software was Bell Northern Research's Virtual Graphics Machine (VGM), which was donated by that company to the Naval Postgraduate School for research purposes. A detailed report on the system installation is presented in Appendix A.

An intermediate goal of the project is to extend the newly installed CORE system to a variety of devices. Toward this end the initial steps were taken to incorporate an interface with the Ramtek RM-9400 graphics display system into the CORE software. Appendix B describes this portion of the research.

As part of building a knowledge base for later research, the CORE system and its development are discussed in detail. The discussion is intended to give enough of an overview of the system so that the reader will not need to refer to the source documents except for essential details. An examination of the relationship between VGM implementation and the CORE specification is also provided.

II. HISTORY

A. THE PROBLEM OF NON-STANDARDIZED GRAPHICS SYSTEMS

Until the mid 1970's individual graphics devices were operated with their own specialized software systems and their instruction sets were tailored to their own particular capabilities. Programming techniques generally were constrained by the device characteristics. Even program structure could be dictated by the available graphics system. Added to these restrictions would be additional requirements associated with installation computing hardware and operating systems. Such highly individualized equipment meant that each graphics system required specialized programs, and, in general, these were applicable for that installation and no other.

As the field of computer graphics expanded, such limitations became a real liability. The inability to use one program at more than one installation meant that for a given application a new set of software would have to be developed individually for each combination of hardware and operating system. The issue of non-portability eventually became an overriding concern of the industry.

If the programs for particular devices were individualized then so was the training of the programmers. Every device required users to have fairly extensive

knowledge of its operation. Rather than concentrate on graphics in a broad spectrum, application programmers got involved in very low level, highly detailed programming for a specific device. This meant that much of the knowledge and techniques developed by a programmer might be unuseable if the device were changed. Thus, a hardware change brought with it the necessity for an in-depth training program on the new device. Further, the programmer would now have to keep the operating details of each device separate in his mind. Confusion of such details is highly conducive to the introduction of additional bugs into programs.

The prospect of working in a very restricted environment and of being required to assimilate and differentiate a host of idiosynchrasies, mnemonics, formats and operational details no doubt caused many potential graphics application programmers to turn to other fields of specialization. The graphics industry must count this to its detriment.

Another problem, perhaps not quite as visible as the portability issues, but certainly as worthy of concern, was the difficulty researchers in graphics had in building on one another's work. An application written, for example, for a storage tube display, would have to be completely rewritten for a raster device. The changes would be so numerous that equivalence of the programs would be impossible to assert. Also, the full duplication of a piece of work simply to adapt it to a different environment was a

costly process in terms of time and resources, both human and machine.

B. FORMATION OF THE GSPC

In more recent years, there have been some attempts to remove the user from the details of device operation by providing high level software. This typically took the form of a package of subroutines callable from some standard high level language. Such packages did remove the need for programmer knowledge of some of the device operating details (though by no means all of them) but each package was still unique. Thus, though a step forward had been made, programs written from different graphics packages were still not portable.

The next step in the evolution was to develop graphics packages that were still specific for particular graphics devices but with a standard interface to the user program. This would allow transportation of user programs unchanged to installations where the "standard" interface was implemented. This development was moderately successful but by this time, areas of research had grown up around particular classes of graphics devices. User's of the various classes of devices all had their own idea as to exactly what form the application program interface should take. These opinions were widely varying and as always, were oriented toward the device capabilities.

The various "schools" of graphics research eventually realized that there were wide areas of agreement in their concepts of how a user should see a graphics system. This led to the next evolutionary stride, the standardized graphics package. These types of systems were designed so that not only would the application program/package interface be fixed, but the functionality of the package itself would be unchanging. The need for individualized software for each particular device, however would not be removed. This task would be accomplished by a "device driver" and its interface with the standard package would be a fixed entity.

The first meaningful steps toward a standard graphics package was the IFIP Working Group 5.2 Graphics Subcommittee's Workshop on Graphics Methodology held in Seillac, France in 1976. Based on experience with existing machine and device independent packages like GINO-F and GPGS the subcommittee laid the groundwork for the movement toward industry wide standardization. Their contribution was to outline and define the issues that had to be addressed if a standard was to become a reality. As already discussed, their prime concern was the issue of application program portability. Toward this end their recommendation was that a study of the structure of application programs was indispensable, and that the results of such a study would drive the specification of a graphics standard. Another

outcome of the Seillac meeting was the assertion that the separation of operations concerned with creating a picture of an object from those concerned with manipulating the object itself was essential. Lastly, the Workshop urged that a standard graphics system specification not stand alone. Along with the detailed design should be included the methodology behind its development.

In 1977 the ACM Special Interest Group for Graphics (SIGGRAPH) appointed a Graphics Standard Planning Committee to design an industry wide graphics standard. Using the recommendations of the Seillac Workshop as a foundation, the GSPC designed the CORE graphics system. Their specification for the system, and the methodology that led to it, were published in August 1977. An updated version appeared two years later incoroporating the experience gained in the interim and extending CORE to raster devices.

III. ISSUES CONSIDERED IN CORE DESIGN

In the early stages of discussion, the Graphics Standards Planning Committee concentrated on clearly defining their goal. Their objective was to design a general purpose graphics system that would meet the needs of the majority of graphics programmers and would be simple to install on most existing interactive displays.

It is essential that such a general purpose system be simple. This principle was recognized by the GSPC and strictly adhered to. They targeted their efforts toward the potential users with the intention of promoting well-structured, comprehensible software. Syntax was considered a less complex and secondary design issue. The committee also sought to put definite bounds on the scope of the new system. The intention was to provide a system that would offer a wide range of capabilities, but not at the cost of losing its appeal as a general purpose tool. Two tradeoffs that constantly cropped up before the GSPC were simplicity vs. wide applicability and program portability vs. machine efficiency.

A. FORMAT OF CORE

The GSPC considered three approaches to development of the core system. One possibility was to create a complete

new graphics language. A second choice was to take an existing language and make the core system an extension of it. The third approach, and the one finally settled upon was to build a package of graphics subroutines. There are a number of advantages to the subroutine package as opposed to either of the alternatives. The major benefit is that it requires no changes to either the language or the compiler. System development, revision, and experimentation will have no effect on any other software at the installation. The primary limitation is that the syntactic structure is extremely limited since the only choice in this area is the approach to parameter passing.

B. DEGREES OF PORTABILITY

The degree of program transportability as measured by the type of required changes was broken down into three general categories:

1) The absolutely portable program which when transferred from one installation to another would require no changes whatever to the source code. Such programs are the ultimate goal of any graphics standard.

2) Programs that require only editorial changes without modification of structure. This class of program would require adaptation to a new installation in much the same way that non-graphics programs written in a high level

language have to be modified when they are transported. The changes typically are those necessary to adapt to such local idiosynchrasies as different character sets. These changes do not require a graphics programmer or a specialist in the particular application. Many of them can even be automated on a reasonably good text editor.

3) The least portable category of programs would be those where changes to the program structure itself are required. Such changes as having to create a particular routine to draw an arc to replace a single command required by a more powerful device fall into this category. Changes of this type may require a detailed knowledge of the particular installation characteristics; even then they can be somewhat difficult and have a tendency to be error prone.

Realization of the absolute portability goal, was not expected when the GSPC published their proposal. The immediate hope for the CORE system is that it will drastically reduce the number of changes required in an application program when it is transported. It is anticipated that the "routine" changes of category 2 above will be required in source code but, as a minimum, the structure of the source program should remain intact.

C. SCOPE OF CORE

Having established the form the roposed graphics standard would take, and the results that could be expected

from it, the committee set about defining the scope of the project. Recognizing the Seillac workshop's wisdom, the GSPC focused on "viewing" as the essential part of any graphics system (its core, hence the name) and treated anything not concerned strictly with displaying information about an object as outside the scope of the standard. This is not to say that such issues as modelling or graphic arts were ignored. The intent was to design a core viewing system upon which those functions dealing with abstract objects and relations between them could be built.

D. VIEWING SYSTEM CONCEPTUAL MODEL

Once the scope of their task was defined, the committee developed a model that would adequately represent their concept of the viewing system of the standard. The viewing system can be thought of as a "synthetic camera" positioned and oriented in a user defined "world coordinate system". The display on the output device would be a "snapshot" of whatever was in the camera's field.

Such an analogy fit well the rules the GSPC had formalized. The "graphical world" was considered all of the graphical data available to the system. What would show up on the output device would be a view of this world through the eye of the synthetic camera. The camera might be able to take in the entire set of graphical data or only a portion of it, depending on the viewing parameters. To take an

imaginary snapshot, the system would have to be aware of the camera's location in space, its orientation, and the amount of a particular object it could actually photograph (i.e. its clipping volume). Further, at the instant the snapshot is taken, none of the parameters related to the object being photographed could change. It must be remembered that what is on the screen is only a part of the total information in the graphical world.

E. GRAPHICAL DATA STRUCTURE

Besides deciding how to display an object in the graphical world the committee had to establish the particular structure to be used to represent graphical data. The simplest approach would be to have no structure at all. Either all of the graphical data is present and fixed or a new graphical world will have to be created. Thus, one could only build or erase an entire object. Once displayed, there would be no changes to the graphical world. Such an approach is very easy to implement and storage is not a major issue, since after the snapshot is taken, there is no further need of the data. Such a system is ideally suited to hard-copy plotters and similar devices but its drawback is that it does not meet the needs of a large portion of graphics applications.

Much of graphics is concerned with building an object and selectively modifying only parts of it. This requires

first, that the graphical data not be lost after it is displayed and second, that it be "segmented" in such a way that individual pieces can be manipulated. Since there is a considerable demand for such structuring, the GSPC incorporated it into their standard proposal. Their concept is that graphical primitives (line drawing, text, and marker placements) be grouped into indivisible and unchangeable segments. These segments are then be combined to produce an entire object. Thus the picture can be modified by pieces through the deletion and addition of individual segments.

In the interest of economy and flexibility the CORE system also implements a form of the simpler unstructured option. For the body of users concerned with storage efficiency or having hardcopy displays, it is possible to designate segments as temporary. A temporary segment is one that would generate graphical output while it is open, but once closed the segment would be automatically deleted. The next "new frame" action will cause it to be lost altogether. Effectively, while the image is being created in the open temporary segment the graphics system is unstructured.

Another possible structuring of graphical data would be to use multi-level units to define the graphical world. In such a system, a "unit" (analogous to a segment) would be a collection not only of primitives but also of references to other units. This approach was considered by the GSPC to be

to complex for the bulk of current graphics applications. There are many variations to its implementation, none are easily accomplished, and all require a great deal of bookkeeping overhead since a change to one unit may ripple through several others that reference it. It should be kept in mind, however, that if such a graphical structure is desired it is possible to construct it using the CORE system.

F. ATTRIBUTES

With the structure of graphical data settled upon, the need arose for defining modifications to the described object. Besides the primitive functions already mentioned, a graphics system must have a means of further describing them. For example, a primitive like "line" might have characteristics such as dashed (style), red (color), and double thickness (width). Deciding how to incorporate such attributes into CORE was another major issue facing GSPC. Some attributes naturally associate themselves with primitives, like line style and width, while others just as naturally only apply to whole segments, such as viewing angle or clipping volume. A problem arises with attributes that are likely to be needed for both primitives and segments. Color is a typical example. The ability to produce a drawing using lines of several different colors might be desirable. But if it is desired later in the program to

change the color of all drawings to say blue, an ambiguity arises in how to handle the multicolored segment.

The GSPC felt that resolution of such ambiguities within CORE would take away from the simplicity of their system. Therefore they established the rule that no attributes would be shared. Attributes would be specific for primitives only or for segments only. The former would be "static" attributes and be unchangeable for a primitive once declared. Segment attributes, on the other hand would not be so restricted. These "dynamic" attributes would be changeable to meet the user's needs at any particular place in the program.

The decision as to just which attributes would be static and which would be dynamic were based on two criteria. The first being that primitive attributes would be those things that would normally be recorded by a snapshot of a particular object. Attributes pertaining to the image as a whole would be segment attributes. The second criteria was that an attribute would be dynamic, (i.e. a segment attribute) only if most medium performance, refresned display device architectures supported changing the attribute with reasonable ease and efficiency.

G. TWO AND THREE DIMENSIONAL GRAPHICS

Along with the questions of graphical data structure, the issue of how to treat two-dimensional and three-

dimensional graphics in a single standard had to be decided. Initially the inclusion of three-dimensional graphics was in question since it necessarily would add complexity to the system. It was decided that the need for three-dimensional graphics was great enough that its exclusion would restrict the applicability of the standard to an undesirable degree.

A more subtle question than 3D inclusion was whether to treat 2D graphics as a subset of 3D or to handle the two as disjoint sets of operations. The advantage of disjoint treatment is that the 2D expression would not be unnecessarily complex since 3D information would not have to be carried with them. On the other hand a large portion of the system's routines would have to be duplicated, one group specialized for 2D manipulation and another for 3D.

The GSPC chose the subset approach in the interest of a unified treatment for all images. To foster simplicity in 2D graphics, they established that the z axis coordinate would automatically default to that of its last specified value when fitting 2D operations into the 3D format.

H. VIEWING TRANSFORMATIONS

One of the most difficult issues for the GSPC to decide was how to treat viewing transformations of an object. There are a large number of approaches to this problem and all have a certain amount of merit. When considering this particular question the committee chose to aim for maximum

system generality. Toward this end, they established four criteria. The first was that any viewing transformations to be performed would be declared before description of a particular object. No transformations within a segment would be allowed. Secondly, two-dimensional transformations would be upward compatible to three-dimensional ones. Third, all general planar projections would be possible to implement. Fourth, parallel and perspective projections would be consistent.

In the discussion of viewing it is necessary to go back to the synthetic camera model. In order to maximize generality of the viewing aspect of CORE the parameters under investigation were those concerning the location of the synthetic camera in space and the location of the viewing plane. The orientation of both the camera and the viewing plane must also be considered. It should be apparent that the most flexible viewing system is the one that allows any orientation and any location for both the camera and the viewing plane. Restricting the positioning of one or both, limits the allowable viewing pyramid and results in failure to meet one or more of the established criteria.

For example, suppose the view plane were restricted so that it must always be normal to the direction in which the camera is aimed. In such a case, oblique perspectives are no longer possible, which is a violation the third criterion.

I. LEVELS OF CORE

One of the final issues to be resolved was the structure of the CORE standard itself. The question was whether to establish a single monolithic standard or to allow a number of "standard subsets" of a parent system. Realizing that a very extensive system might be well beyond the needs and/or resources of many installations, the GSPC settled on a three level structure. The lowest level would be restricted to the most basic needs of most users, stressing ease of implementation as well as economy of computing resources. The upper two levels include more features and a higher capability with correspondingly more difficulty in implementation. Each upper level of CORE includes all of the features of the level below it.

IV. CORE SYSTEM DESCRIPTION

A. OVERVIEW

In the previous chapter most of the basic terminology of CORE has been introduced. In this chapter the system itself is discussed in detail, but before doing so, more terminology must be introduced. The information displayed on the graphics display device is referred to as a *picture*. The basic building blocks of pictures are *output primitives*. An output primitive is a line or sequence of lines, a non-drawing move of the cursor, a marker placement, or a string of text. A number of output primitives are grouped together to form a *segment*. Each segment defines a single *object* and a combination of one or more objects defines an *image*. The *view* of an image can be thought of as a imaginary camera snapshot of it. To obtain a 3-D projection of an image the user specifies the imaginary camera position, type of projection (perspective or parallel) and where on the display surface the object is to appear. Different views are obtained by "moving" the synthetic camera in space relative to the stationary object. After the view of an image is determined the graphics system must map it onto the particular device selected to show it. The CORE system does this using two coordinate systems. Objects and images are created in a user defined, previously specified *World*

Coordinate System. Within this coordinate system the part of the total image that is to be displayed is framed by a window. The World Coordinate System is mapped by CORE onto a set of normalized device coordinates (NDC). NDC specification defines the viewing area on the selected graphics device that will be used. NDC's are specified as fractions of the total available display width and height. The window and the visible section of the image in it are mapped to the corresponding location in the normalized device coordinates. Once the image is in in terms of the NDC it is a simple matter for the CORE system, knowing the particular device characteristics, to translate it into a picture on the screen.

Any graphics system must have a means of controlling its operating environment. In the CORE system this is accomplished by:

1) turning clipping on or off

2) selecting view surfaces for output

3) setting initial values for segment and primitive attributes.

4) establishing error handling mechanisms

To support the control functions the application program is given the capability to inquire about the system status, variables and device capabilities. There is a "new frame" function for screen clearing and a capability for grouping changes to the picture.

Output primitive functions may be referenced either by a segment identifier or a special "pick-id" name. The pick-id is used in conjunction with a pick device, which will be discussed later in the chapter.

To allow utilization by the system of specific hardware and installation features, there is an escape mechanism. It is a standard, rigorously constrained function that allows the CORE system to take advantage of the non-standard capabilities of its environment. Use of the escape has a price in that it takes away from the portability of the application program.

B. OUTPUT PRIMITIVE FUNCTIONS

Output primitive functions are the operations the programmer uses to describe objects in the device-independent World Coordinate System. Invocations of these functions are gathered into segments as drawing commands. Primitives work from the current position (CP), which is a drawing location in world coordinates. It is simply a starting point for application of the function, and is initialized to the origin upon segment creation.

There are five output primitives: single and multiple line drawings, text display, and single and multiple marker placements. These are only slightly different for the two- and three-dimensional versions. Coordinate positions may be specified as either relative or absolute, but the former is

merely a notational convenience. It does not necessarily generate a relative positioning command to the hardware. The concept of a marker in the CORE system is simply a designation of a position in world coordinates. A particular character appears on the view surface to indicate this position but in world coordinates there is no such character.

Three kinds of text are supported by the CORE system output primitives: string precision, character precision and stroke precision. The main purpose of string precision text is to supply information to the operator. Its generation is simple and efficient. Character precision text is used when it is important that a character string occupy a designated space, a plot axis, for example. String and character precision are also referred to as low and medium quality text, respectively. Both medium and low quality text output primitives take advantage of hardware character generators, if available. Stroke precision, or high quality text requires a different approach. Here, the string is treated as if each line of each character were generated by software in the CORE system.

C. SEGMENTS

Segments are created in the applications program. Creation of a segment follows a simple sequence. The World Coordinate System is defined and normalized device

coordinates are specified. If desired, the synthetic camera, discussed earlier, is positioned to establish the view of the object. Next, a segment is "opened" and the object described using the output primitives. After completing the object description, the segment must be "closed ". To modify a piece of the picture a segment is deleted and a new one created to replace it. Segments are of two types: 1) retained, which are typically used for buffered displays and 2) temporary, which are most often used by plotters. As one might infer, temporary segments are used only once to create a display and then are discarded. Retained segments are kept by the CORE system until specifically deleted. Temporary segments have the advantage of economy of memory utilization. A segment's type is established when it is created and remains unchanged for the life of the segment. Copying one segment into another or invoking one segment from another is not permitted under the CORE system.

D. ATTRIBUTES

The effects of output primitives are modified by assigning attributes to them. For example, the primitive "line" has an attribute "linestyle" which has values "solid" and "dashed". Other attributes that apply to primitives are color, character size, character precision, linewidth and more.

Segments, like output primitives, also have attributes. These control such things as a segment's visibility, highlighting within the segment and its detectability by a pick device. This last is a particularly important segment attribute. When a segment is detectable and a pick is enabled, the device can select a primitive from the segment and return to the application program both the segment name and the primitive's pick-id.

With the exception of type, segment attributes are all "dynamic" in that they may be changed after the segment has been created. If the user does not specify attribute values prior to segment creation, the CORE system provides a set of default values.

Segments are assigned attribute values from a table of current attribute values maintained by the system. The application program has the capability to interrogate and change attributes. For primitive attributes changes can only be made while the segment is open; segment attributes, on the other hand, may be changed at any time. A single attribute cannot apply to both primitives and segments. If certain attributes are not supported by hardware, the options are to either simulate them or force a reference to an error handling routine. The choice is installation dependent.

Besides segment and primitive, attributes can be classified by the "space" in which they operate. For

example, text attributes describe the text regardless of its location or orientation. They are said to define characteristics in "object space". On the other hand, line attributes such as style and width are related to views of objects. Depending on the location of the synthetic camera these attributes of an object may appear different for the same value. They are said to operate in the "picture space".

E. VIEWING TRANSFORMATIONS

A viewing transformation accomplishes two tasks: it specifies how much of the world coordinate space is visible and it maps visible world coordinate pictures into normalized device coordinates. The viewing transformation takes a world coordinate volume (a clipped portion of a complete display) and projects it onto a view plane in world coordinates defined by a window. The projection is then mapped into a normalized device coordinate viewport, and finally to the physical device coordinates. The core system avoids a problem that has occurred in the past where two-dimensional objects required different treatment. 2D objects are treated as a subset of the 3D objects. When a Z component is not specified, a default to the Z component of the current position is effected.

Window rotation or inclination is a common requirement for many applications. In the CORE system the concept is implemented using a view-up vector. This vector simply

points to the "straight up" direction for the window with respect to the world coordinate orthogonal X, Y, and Z axes.

F. INPUT PRIMITIVES

Six types of input devices are supported by the CORE system:

PICKS: identify an output primitive by its segment name and pick-id.

LOCATORS: provide world coordinate values for a position on the view surface.

VALUATORS: provide a scalar value.

KEYBOARDS: provide character strings.

BUTTONS: provide a means of selecting from several alternatives.

STROKES: provide a series of positions to the application program in normalized device coordinates.

Input for interactive graphics is accomplished through logical input devices. These devices are a specified abstractly in the application program. The program defines and controls them in a way unaffected by the hardware. The CORE system's task in interactive graphics is to connect logical input devices to an available piece of hardware that will accomplish the desired function. Logical input devices may be manipulated in the following ways:

1) Initialization/ termination

2) Enabling/disabling

3) Event queueing/ dequeueing

4) Sampling

5) Associating sampled and event causing devices (this ties values provided by sampled devices to events caused by event-generating devices)

6) Echo control

Logical input devices fall into two mutually exclusive categories. They are either sampled devices or event causing devices. Stroke, pick, keyboard and button are event generating devices; locator and valuator are sampled devices. Event-causing devices provide signals to the application program. For each event, an event report is created containing data related to the state of the device at the time of the event. The CORE system enters event reports in an event queue for later use by the applications program. To get state information about sampled devices, the application program must query them. These devices do not generate event reports. A standard feature of the CORE system is to echo automatically all operator interactions unless this function is specifically deactivated.

G. LEVELS OF CORE

To meet the wide range of installation capabilities and requirements, an upward compatible three-level structure for the CORE system was selected. The aim was to accomodate what were considered the most common classes of equipment and

applications. The most basic level of the CORE system deals strictly with output. There is no interactive capability and the segments are of the temporary type only. This level consists of the output primitives and their attributes, viewing transformations and device controls.

The next level adds the ability to retain selected segments. It still is limited to output only operation. The visibility and highlighting segment attributes also are included. The third, dynamic level, allows use of the input capabilities. This is the level at which interactive graphics is supported. It provides all the functions intended to make up the complete core system:

1) Output primitives and their attributes

2) Viewing transformations

3) Device control

4) Temporary and Retained segments and their attributes

5) Input primitives

6) Image transformations

Level three is further divided according to the capability for image transformation:

3A) Two-dimensional translation only

3B) Two-dimensional translation, rotation and scale

3C) Three-dimensional translation, rotation and scale

As with all of the levels, these sub-levels are also upward compatible.

Complications with such a level structure are likely to arise at installations where there is a combination of different graphics devices. What is envisioned for such a facility is a body of device independent code linked to individual device drivers. The intent is to share as much of the independent code as possible, thereby keeping as much to the objective of probability as feasible.

H. ENVIRONMENT INTERFACE PROBLEMS

Despite a great deal of effort to make the CORE system a stand-alone entity, operating systems and programming languages still impact upon it. For instance, there is no standard way to make device driver routines available to the application when the system is invoked. Methods can vary widely depending upon computer and operating system capabilities. Another problem is the case where a system message is sent to a terminal where the CORE system has been invoked. The state of the display may be changed without the system being aware of it. The consequences of this depend on the situation, but system reliability will certainly be degraded.

There is as yet, no definition of a standard interface with programming languages. It is hoped that as more insight and experience is gained, a standard language interface will be developed and the CORE system will be able to be invoked from more than one language, adding a new dimension to its

portability. Input/output also has problem potential if the programming language and the CORE system are operating on the same device. Resolution of this is still highly language and device dependent.

V. THE VGM APPROACH TO CORE

The Virtual Graphics Machine (VGM) is Bell Northern Research's implementation of the CORE graphics system. It was developed on an IBM 3033 in ANSI standard FORTRAN and later modified to operate on a PDP-11/70 under the RSX-11 operating system. VGM is a FORTRAN based set of subroutines with each subroutine corresponding to a CORE primitive invocation, attribute setting, or view transformation. The package also includes subroutines for control purposes, such as initializing devices, opening segments and setting up coordinate systems.

The intended customer market for VGM is installations with low and medium cost "intelligent" terminals which are capable of generating graphical output from fairly high level functions and primitives. Terminals not accepting such high level input will require intermediate software to either simulate the functions or break them down to lower level primitives compatible with the device.

Under RSX-11, VGM exists as a library of FORTRAN subroutines. To use VGM, the application program is created independently as a main program making calls to the VGM library. The application source code is then compiled independently. The connection with VGM is made by the RSX-11 Task Builder. The application object file and the

appropriate routines from the VGM library are linked into a single task by that RSX utility.

Included in the VGM library is the particular subroutine that establishes communication between VGM and the selected device. This segment of code has to be created specifically for the installation where VGM is to be implemented. This routine, SELSTR, is the only executable code in VGM that interfaces with the device driver. Graphical data is passed between VGM and the driver via a COMMON block of memory. What SELSTR does is set the necessary flags to control the concurrency between the application task (linked with VGM) and the selected device driver tasks. Each device driver exists as a separately compiled and linked task. Under RSX, before invoking any driver from VGM these tasks must be INSTALLed by the user.

In VGM, syntax is a very minor issue. Since the parent language is FORTRAN, and the entire system is based on the subroutine call, the only syntax is the manner in which the necessary parameters are passed.

It should be emphasized that VGM does not implement the CORE graphics system exactly as set down in the 1979 GSPC report. There are a number of differences, which may be grouped into four general categories.

A. FUNCTIONAL DIFFERENCES

In this category the end result of a series of operations in VGM is the same as that specified by CORE but

the mechanism for achieving the result is not the one specified in the proposal.

1. Segmentation

The CORE system creates a retained segment or a temporary segment with a single function specific for the particular segment type. VGM uses a two-step process. First, a segment type is established. After the invocation of the routine to do this any segment created will take on the type of the one declared. All segments created will be of the same type until a new type is declared.

Retained segments in VGM are stored in Transformed Display Files (TDF's). The TDF contains graphical information that is ready to be translated into a device compatible format. All clipping and transformation has been done before the data is entered in the TDF. Should the application program specify an operation on a segment, the entire TDF is likely to be changed. Segments that are specified to be temporary do not cause creation of a TDF.

2. Attributes

Like CORE, VGM partitions attributes according to their application to either segments or primitives. Both systems further divide the set of attributes according to their changeability within the program, dynamic attributes being those that are subject to change by the application program after their initial declaration and static

attributes being those that are not. In CORE there are no dynamic primitive attributes. VGM, however, does have a group of primitive attributes that it labels as "dynamic". Each member of the set of VGM dynamic primitive attributes is also a member of the set of static primitive attributes.

In VGM a static primitive attribute is one that is set while a segment is open, and that once declared, applies to all appropriate primitive invocations following it until the segment is closed. Further, for the lifetime of that segment it will always apply to the set of primitives created with it. A static primitive attribute cannot be overridden by any other setting of that attribute anywhere in the program.

If no attributes applicable to a particular primitive are set within a segment then dynamic primitive attributes may be assigned outside the segment. At a later time in the program these attributes may be changed. When a dynamic primitive attribute is set, the segments to which the change applies must be specified. If the application program does not set either dynamic or static attributes for some primitives then default values are used. It is also possible for the user to specify his own set of default values.

The applicability of an attribute to a primitive at any particular time in the program can be determined by the rule that user specified dynamic primitive attributes always

override default values and static primitive attributes always override dynamic ones.

3. The Viewing Surface

The flow of graphical information from a segment to an output device is viewed by VGM as a "stream". By manipulating streams, the user carries out the CORE SELECT/DESELECT and ENABLE/DISABLE device operations. In VGM when the user initializes a stream, he is picking a particular device or group of devices for output. Devices are assigned to a specific stream as part of VGM. A given stream may have more than one device associated with it. Changing this assignment requires changes to the VGM source code itself. Stream initialization by the user's subroutine calls accomplishes the necessary operating system functions to link VGM and the appropriate drivers. It is valid for more than one stream to be in use at any given time.

After initializing the required treams the user then selects one or more of them to be used for display. Once a stream is selected, all subsequently generated graphical output will be displayed on the devices assigned to that stream until it is deselected. There is no way to address individual devices on the same stream. Stream operations are not allowed while a segment, regardless of type, is open.

4. Coordinate Systems

VGM uses an extra coordinate system in translating graphical data from world coordinates to the terminal Physical Device Coordinates (PDC). Between the transformation from World Coordinates to Normalized Device Coordinates, VGM takes clipped graphical data and maps it onto a view plane in View Plane Coordinates (VPC). The flow of graphical data is shown in Figure 1.

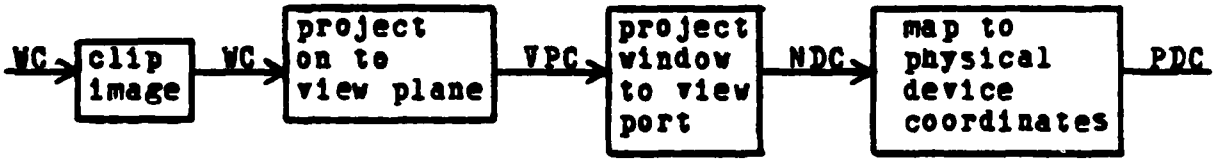


Figure 1. Flow of graphical information through coordinate systems

5. Transformations

In the CORE specification there is a static segment attribute called IMAGE_TRANSFORMATION_TYPE this specifies a maximum allowable level of transformation that can be applied to a given retained segment. There are four allowable levels:

a. no transformation

b. 2D translation only

c. 2D translation, rotation and scale

d. 3D translation, rotation and scale.

This feature is not included in VGM. The transformations of 2D or 3D translation, rotation and scaling may be applied to any retained segment at any time in the program.

6. Text Manipulations

In CORE, the manner in which text is displayed on a device is controlled by, among others, the attributes CHARPATH, CHARJUST, and CHARUP. The first attribute specifies one of four paths in the view plane: up, down, right or left. As the sequence of text is output CHARPATH determines where in relation to the last character the next is to be positioned. The first character is always positioned at the CP. The CHARJUST attribute is a combination of directions, again in the view plane, which indicate where, in relation to the CP, the rectangle defined by the output text string is to be placed. Figure 2 gives the possible CP locations.

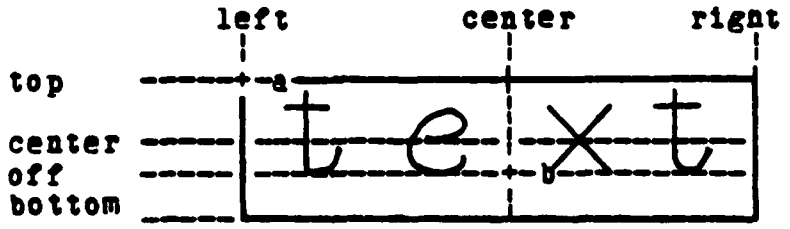
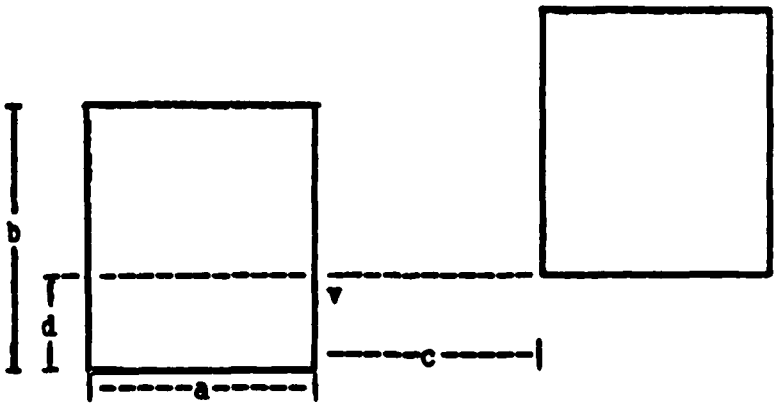


Figure 2. Possible position designations of CHARJUST

The text, depending upon the charjust values will be placed in such a way that the CP will be at a junction of a

vertical and a horizontal line. A particular junction is identified by its horizontal and vertical position labels, e.g. "left, top" = point a; "center, off" = point b. The CHARUP attribute is a vector from the origin in World Coordinates which specifies the "up" direction for the text.

These three CORE attributes are not specifically implemented in VGM. Instead, their functionality is included in the VGM attributes CHARPLANE, CHARSIZE and CHARSPACE. The text string orientation is defined by the CHARPLANE (a vector in World Coordinates originating at the CP) and a "string extent" vector. The string extent vector is obtained from the CHARSPACE and CHARSIZE attributes. Figure 3 illustrates the components of these two attributes.



a = CHARSIZE x component
b = CHARSIZE y component
c = CHARSPACE dx component
d = CHARSPACE dy component

Figure 3. CHARSIZE and CHARSPACE attribute components

The string extent is the result of multiplying the vector 'V' by the number of characters. 'V' originates at the CP. The boxes containing the text characters will be in the character plane with their lower left corner on the string extent vector. A wide variety of directions the text may follow stems from the fact that values for the attribute components can be positive or negative.

7. Visibility

In CORE there is a segment attribute called VISIBILITY which, if "on", means to display a specified segment on the output device and, if "off", to remove it from the screen. This capability also exists in VGM where a segment may be POSTed to make it visible or UNPOSTed to remove it from the picture.

B. CORE FUNCTIONS NOT IMPLEMENTED BY VGM

The following list of CORE functions are not implemented at all in VGM:

1. pen attribute
2. marker_symbol attribute
3. pick_id attribute
4. naming of primitives
5. view_up vector
6. some inquiry routines
7. terminate_, disable_, and enable_group routines
8. batch update

9. escape mechanism

10. highlighting

11. hierarchical level structure

12. acceptance of asynchronous input

C. FUNCTIONS IMPLEMENTED BY VGM NOT IN CORE SPECIFICATION

1. Primitives

The following primitives have been added to VGM:

a. rectangle

b. arc

c. polygon

d. flood

The *flood* primitive is for use with bit mapped, color devices. Flood locates the CP and establishes the smallest area surrounding it bounded by arcs or lines. This area is then filled with a user specified color. If the CP is not enclosed then it is possible to flood the entire display surface.

2. Attributes

For terminals capable of color graphics VGM adds a BACKGROUND_COLOR attribute and an ADDITIVE_MODE attribute. The former is self explanatory. The latter determines how a declaration of any new color attribute is to be treated. If ADDITIVE_MODE is "on" then the bit pattern for the old color is logically ORed with the bit pattern for the new color,

and the resulting pattern becomes the value of the attribute. Otherwise the new color bit pattern simply replaces the old one.

A BLINK attribute is implemented in VGM, which is intended to be one method of replacing the CORE system HIGHLIGHTING attribute which has been left out.

3. Other Features

In VGM, there is a mechanism for modifying a segment after it has been closed. This is the EXTSEG feature which effectively re-opens the segment and allows additional graphical information to be appended to the existing file. The feature only allows addition of information and requires that the CLOSEG command be issued after the addition is complete.

If a graphics device that is currently in use has both input and output capabilities, VGM will, if directed by the application program, "back transform" input coordinates from Physical Device Coordinates to World Coordinates. CORE will only back transform to Normalized Device Coordinates.

VGM's error handling and debugging aids offer more than is required by the CORE proposal. In VGM the user has the capability to specify the maximum tolerable error severity and the maximum tolerable number of errors. If either maximum is exceeded, the program will terminate. When errors are detected, an entry is made into an error trace

file. This file is intended to be a debugging tool. It contains the error code number, a brief description of the error, the relevant parameters involved in the error, the name of the routine in which the error was detected and the result of the error (corrected, ignored, default substitution or program termination).

An option that GSPC left open to implementors was how to treat non-graphical data sent to a terminal being used for CORE graphical output. Typically, this might be parent language I/O in the form of write statements or a system message to the particular terminal. In VGM there is a SET_POSITION function which identifies an NDC position specifying where non-graphical output is to appear on the screen. This output is affected by neither attributes nor the CP.

D. EQUIVALENT FUNCTIONS WITH DIFFERENT NAMES

This is the simplest category of differences between CORE and VGM. Below is a short list showing equivalent functions in CORE and VGM.

CORE name	VGM name
charprecision	charquality
detectability	pickability
highlighting	blink
world coordinate transformation	modelling transformation

VI. THE VGM DEVICE DRIVER

The device driver is the connecting link between VGM and graphics hardware. Its purpose is to take graphical data from VGM via the designated COMMON storage area and construct an instruction in a format compatible with the particular device it is written for. The software for the device driver is divided into 6 modules.

A. THE DEVICE CHARACTERISTICS TABLE

This table is a COMMON block of variables describing the characteristics of the graphics device for which the driver is written. It is implemented as a BLOCK DATA source program and is accessed by all of the executable modules of the driver. It is initialized when the module is compiled and is treated as "read only" by all of the routines referencing it.

B. THE STREAM INFORMATION TABLE

This is another COMMON block of data which holds the current value settings for attributes for each stream. The table is updated by the driver routines as the values are changed. When the BLOCK DATA source file is compiled, the default attribute values are set.

C. THE RUN TIME INFORMATION TABLE

This table, like the stream information table is subject to continuous update by the device driver routines. It contains the buffer that holds the instruction to be sent to the graphics device. Pointers required for keeping current positions in the instruction buffer (CODBUF) are also in the run time information table. In addition there are variables for the identification of the debug file and several host computer related values.

D. ROUTINES EXECUTING VGM PRIMITIVES (OnLIB1)

This module is executable code that is intermediate between VGM and the device instruction creating portion of the driver. Its routines are invoked from VGM and it in turn calls routines to create the appropriate data to fill CODBUF. OnLIB1 is graphics device independent but is host machine dependent. Contained in OnLIB1 is the OnEXEC routine that is the only executable code in the driver software that communicates with VGM.

E. DEVICE INDEPENDENT LIBRARY OF SHARABLE ROUTINES (SKELIB)

This collection of routines is a set of device independent operations that are optionally available to OnLIB1. These routines perform operations like projection on a plane, clipping, image transformations, line style generation etc. For highly capable devices which perform these tasks themselves OnLIB1 would not reference SKELIB but

instead cause the specific device instructions to be generated. For devices that lack some of these features in hardware, SKELIB provides software simulation.

F. DEVICE DEPENDENT ROUTINES GENERATING INSTRUCTION CODES (OnLIB2)

This set of subroutines fills the instruction buffer with instructions and data specifically formatted for the target device. Each byte of CODBUF must be precisely set to be compatible with the graphics device. OnLIB2 builds the full instruction and, when complete, causes it to be sent to the terminal. The communication with the terminal is done with MACRO routine QWRITE which uses the host computer's I/O communication facilities and treats the graphics device as an I/O port.

All of the device drivers share the stream information table and SKELIB. Each driver installed with VGM however must contain each of the other four modules. The interconnection of the various device driver modules is shown in Figure 4.

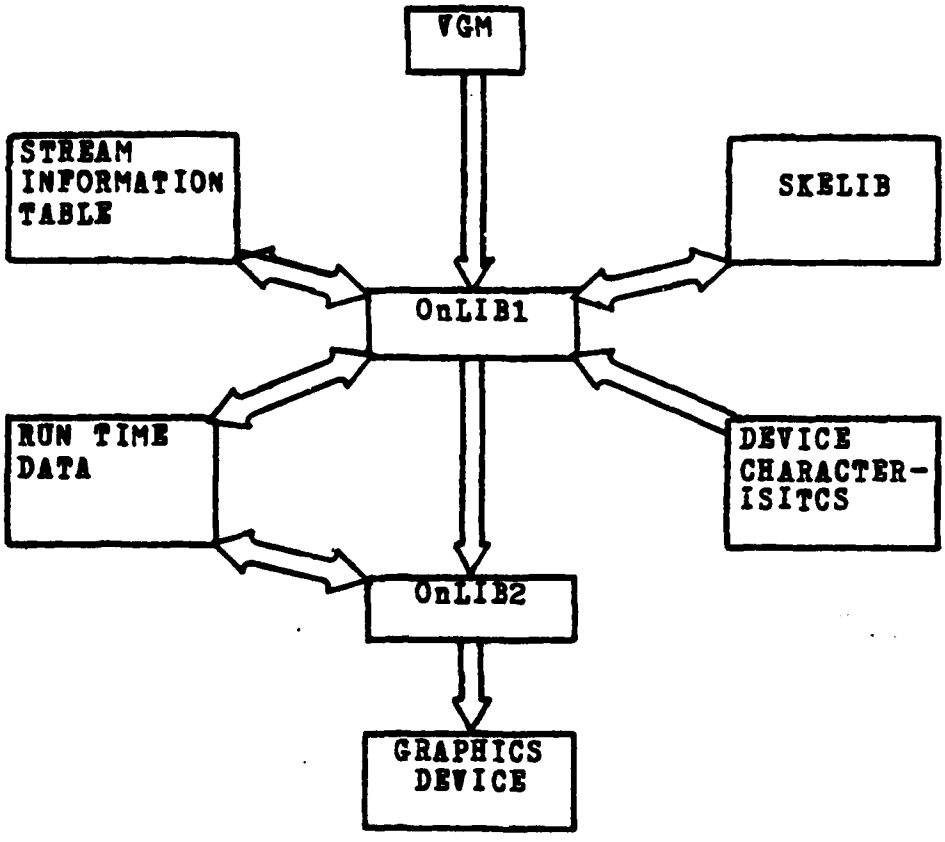


Figure 4. Interrelationship of Device Driver Modules

VII. RESULTS AND SUGGESTIONS FOR FUTURE STUDY

A. SOFTWARE

As stated in the introduction, the purpose of this research was to lay the groundwork for a detailed study of the CORE graphics system. A great deal of progress has been made in this effort. The VGM implementation of the CORE system is installed at the Naval Postgraduate School and is capable of operating with the Tektronix 4014 storage tube display terminal. The system has passed the initial stages of testing. Additionally, algorithms have been developed and implemented on a limited basis for expanding the VGM software to interface with the Ramtek RM-9400 graphics system. These portions of the project are discussed in detail in Appendices A and B respectively.

A less tangible, but equally valuable result of this study is the experience and insight gained with both CORE and the VGM implementation. Appendix C is one product of this new knowledge. It is a brief tutorial on programming with VGM and is written specifically for the Naval Postgraduate School installation. The tutorial is not intended to replace the Bell Northern Research User's Manuals. It is meant to be used in conjunction with them and deliberately avoids details which can easily be found by referencing them.

B. CORE EVALUATION

The CORE system is currently only a proposal and as such it is intended for thorough scrutiny by the graphics community. In researching this subject a number of issues have been identified which may provide a framework for an evaluation of the system. This collection does not purport to be exhaustive but is presented to provide a base for future work.

1. Portability

The prime issue to be evaluated is that of program portability. This has already been discussed in depth in the GSPC proposal and summarized in this report in Chapter III. In the course of this study some different perspectives on the problem have come to light. It appears that there might be hierarchical levels of portability other than those listed in the GSPC report. Graphics devices, computing systems, operating systems and CORE implementations are all variables in this area. Another way of classifying the portability of a program might be in terms of these environmental factors. For example, some programs may be portable from one device to another as long as the computing environment is not changed. Others may survive a change of computing machinery provided the operating systems are the same. Conversely, changing operating systems rather than computers might be the defeating factor in a program's

portability. Yet another possibility is that different implementations of CORE may be the reason for changes in a given program.

It would be difficult to develop further the portability issue from this point of view until a variety of environments exist for side by side comparisons. The presentation of this perspective is recorded here so that when such facilities are available its validity can be considered.

2. Implementation Effort

To gain widespread acceptance the CORE system must be easy to implement. Criteria are needed to measure the difficulty of implementation. The following list presents questions which may serve as possible evaluation mechanisms for this:

a. Can the implementation be reduced to simply following some kind of implementation "algorithm"?

b. Does the design of the implementation favor one type of device over another?

c. Does the design of the implementation favor one computing environment (either hardware or operating system) over another?

d. What tradeoffs in portability have to be made so that implementation can be facilitated?

3. Device Capability

As is true of almost all standards in the computing industry the CORE system does not take full advantage of the hardware capability of many installations. It was never intended to meet all possible needs. Nonetheless a means of assessing the loss in device capability under the CORE system should be established. A guideline for determining when the gains from using CORE are outweighed by the losses from not utilizing the full power of the device would be a highly desirable tool, particularly for facilities generating a wide range of graphics applications programs.

4. CORE Capability

Perhaps the most difficult and controversial question in the evaluation of CORE is whether its capabilities really are sufficient for most graphics programmers. Further, how adaptable to future needs will the system be? Is hardware technology likely to progress to a point where CORE is no longer adequate as a standard? Is it likely that computer graphics will expand into areas that CORE was never intended to serve? Certainly none of these issues will be resolved easily. Each question in itself could be a topic for detailed analysis. They are presented here just to suggest areas for further study.

C. OUTLINE OF CONTINUING DEVELOPMENT OF THE NPS SYSTEM

In the course of this study some ideas have been formulated for a methodology to direct continuing work on

the project. This is only one worker's point of view and it should serve as a guide rather than an absolute to follow-on workers.

1. VGM

The first order of business should be to modify the Bell Northern Research test routine, AKPAK, so that the full range of VGM functionality can be verified. This would entail construction of appropriate overlays for the test package so that the large amount of object code can be accomodated on the limited PDP-11/50 memory. Once this is accomplished, AKPAK can serve as a benchmark program for testing additional drivers that are added to VGM. Using AKPAK as a benchmark should also aid in studying other portability issues.

2. Device Drivers

The pattern for writing the code that interfaces the device independent portion of a software driver and the RM-9400 has been established. Further, it has been partially implemented and tested. The next step is to complete the remaining subroutines in the O4LIB2 (device dependent routines) module according to the algorithms provided in Appendix B. Once a new O4LIB2 has been built it can be incorporated into VGM and tested, first as a separate entity using the provided DDTEST program and then as a fully integrated part of VGM using AKPAK.

By no means would this complete work on the Ramtek driver. After the 04LIB2 work is completed the driver would still fall far short of taking full advantage of the power of the RM-9400. A topic of study all its own would be the modification of the "device-independent" sections of the driver code to put the sophisticated features of the RM-9400 into use.

An ancillary project would be to develop yet another driver for a new graphics device. The object of this study would not be to merely expand the capabilities of the existing VGM system. What it is hoped would emerge from such research is a pattern for writing device drivers. Such a formula, if it exists, would be a very useful tool for further expansion of VGM without the necessity of re-learning already established techniques.

3. Portability

With a fully operational VGM system and a variety of devices available for use, portability testing can begin. The suggestion is to direct the work along lines mentioned earlier in this chapter in section B. After varying the graphics devices and studying the system behavior over a variety of them, work should proceed to studying program behavior under changes in the computing environment. Other operating systems, other computers, and other CORE implementations are available for such research. This

variety of environments in a single location would provide an excellent test bed for thorough evaluation of the CORE standard.

APPENDIX A. IMPLEMENTATION OF VGM AT NAVAL POSTGRADUATE SCHOOL

The aim of this portion of the project was to gain insight into the operation of VGM and to install a working version of it on the Naval Postgraduate School's graphics facility. Bell Northern Research (BNR) of Ottawa, Canada supplied the school with a tape containing the source code for VGM, version 1.1. and two device drivers. One driver was written for a Chromatics color graphics, raster scan terminal and the other for a Tektronix 401X storage tube terminal. A Tektronix 4014 was readily available at Naval Postgraduate School and, being a simpler device, was deemed the best choice for installing and testing VGM.

On the same tape as the VGM and device driver software were several command files to aid in installing the system. These command files did such things as compile the source modules, install libraries and build tasks. Their inclusion was intended to save some user time and effort and to help avoid erroneous or incomplete system commands that might arise during any of the initial stages of software installation.

The plan for implementation was the following:

A. Compile all of the source modules making up VGM.

B. Convert the VGM object code into a FORTRAN library under RSX.

C. Compile all of the source modules making up the Tektronix device driver.

D. Link the object code from the device driver compilations into a single task called O2DRIV.

E. Test the driver separately from VGM by using a test routine supplied by Bell Northern Research.

F. Install O2DRIV under RSX as a task available for concurrent use.

G. Test VGM itself using the Bell Northern Research supplied test graphics program AKPAK.

Before any of the installation could begin, the software on the tape had to be made easily accessible. This was done by copying it onto the RSX on-line disk storage. A copy of the BNR software may be found under directory DP0:[201,211]. This copy of the source code is intended to remain unedited. Any changes to the code during VGM installation were made by transferring original copies to a working directory and editing there. To generate object code the modules necessary for VGM and the device driver compilation were PIPed to directory DP3:[201,210]. Once the source code was available, the command file VGMCOM.CMD was executed. VGMCOM.CMD had to be modified somewhat to make it compatible with the F4P compiler. The BNR software was written under an older PDP

version of FORTRAN so the compiler commands had to be adjusted accordingly.

In all, VGM contains 12 output modules and 7 input modules. Each module is, in turn, made up of several subroutines grouped by the particular function they perform. For example, the output module INISTR (INItialize STRing -- named for the first of the component subroutines) contains 16 separate subroutines all having to do with either stream or segment manipulations. There are also two block data modules and a test program.

Once object code was generated for all of the VGM routines, command file VGMLIB.CMD was executed under the RSX LIBRARY utility, creating a library of all the executable routines concerned with input and output.

The compilation process was repeated for the source code modules for the Tektronix driver. Command file 02DCOM.CMD accomplished this. The device driver modules consist of two "libraries" which manipulate the device independent information coming from VGM. A third library, 02LIB2, does the actual creation of instructions to the Tektronix. 02LIB2 is the portion of the device driver that links VGM and the terminal. There are also some block data modules which set up communication areas between the device driver and VGM and also provide specific device parameters where needed to both VGM and the device-independent portions of the driver. Two MACRO modules are included to handle input and output

communication between the device and the operating system. The rest of the driver related modules concern themselves with interfacing the device independent parts of the driver and VGM itself.

Like VGM, the object code for the device driver modules are built into a library under RSX by file O2DLIB.CMD. This library however is only a temporary holding area for the driver object code. It is referenced by the file O2DRIV.CMD and the object modules are linked together into a single task called O2DRIV.

With VGM existing as an object library and O2DRIV as an individual task the preliminary work is done. Testing is accomplished in two stages. The first is testing the functionality of the device driver independent of VGM. The test program, O2TEST, was provided by Bell Northern Research and was compiled with the rest of the driver routines and block data programs. The file O2TEST.CMD links the test program, the driver library, and the block data into a single executable task called TEKTEST. TEKTEST exercises the driver's functionality fully and is available for use in directory DP3:[201,210].

After the driver was satisfactorily tested alone, the task O2DRIV was INSTALLed under RSX. The INSTALL feature is an RSX utility that activates a specified task and makes it available for invocation from another active task.

The next step is building the VGM test task. The test routine, AKPAK, is also supplied by BNR and is intended to thoroughly test VGM and any selected device driver. The command file provided to build the AKPAK task is VGMTKB.CMD. The initial attempts to build the test program resulted in a Task Builder diagnostic indicating that not enough contiguous disk space was available for the AKPAK task. This was partly a result of the fact that AKPAK and the associated VGM library routines require a very large block of disk storage. This is also indicative of another potential problem. Since the AKPAK test is such a large one it is not likely to fit into the available core memory. It will most likely require construction of overlays to run on the limited memory resources of the PDP-11/50 at NPS.

It was decided that at least for the initial phase of study to follow a somewhat shorter and simpler testing path. Rather than become involved in constructing overlays and possible changes in the structure of AKPAK, a more limited test program would be written. Since one goal of this part of the work was to get an operational VGM system it was decided to at least ensure the proper interfacing between the user program, VGM and O2DRIV.

Toward this end, a much simplified test program was written, called VGMTST. It was linked with VGM and the necessary block data routines by file VTST.CMD. VGMTST exercised several of the 2-dimensional drawing, move, and

marker primitives, the low quality text capability and the SET_POSITION feature. In addition, some of the device and segment control functions were also tested since their use is essential for any VGM operation.

The execution of VGMTST was quite successful. The drawings on the Tektronix screen appeared as expected. The task is available for use under directory DP3:[201,210]. To use the test routine, turn on the Tektronix 4014 and log in under RSX by the standard procedure. Enter the proper directory by typing

SET /UIC = [201,210]

after the RSX prompt (>). Next, when the prompt appears issue the commands

>INS O2DRIV

>RUN VGMTST

The test program will execute and the resultant graphics will appear on the screen.

APPENDIX B. CONSTRUCTION OF A DEVICE DRIVER FOR THE RAMTEK RM-9400

A. DEVICE DRIVERS

Bell Northern Research's device drivers are identified by unique integer designations, the particular integer being incorporated into the driver name and the names of the modules that comprise them. The modules, and subroutines in them, specific to the Chromatics driver all begin with an 01- identifier, for example, 01LIB2 or 01MOVE. The Tektronix 4014 software is device driver #2, thus its names are all prefixed with an 02-. Also provided is a driver which shares the Tektronix 4014 code but is to be used with the Tektronix 4010. This uses the 03- prefix. Only the 03EXEC (the interface with the VGM software) routine has the prefix, the rest of the driver uses the Tektronix 4014 code. For the routines to be written as part of this study, the 04- prefix was selected, being the next in the sequence.

The target device in this portion of the study was the Ramtek RM-9400, a very powerful and sophisticated color raster scan graphics device. Adaptation of a similar driver seemed the best course of action. The Chromatics graphics terminal is also a color, raster device though not nearly as advanced as the RM-9400. Nonetheless its driver is readily available and is already adapted to the PDP-11/RSX

environment. It provides a good starting point for developing the Ramtek software.

The initial intent in the construction of a device driver for the RM-9400 was to leave the Chromatics driver intact as far as possible. The module SKELIB, which contains the device independent hardware feature simulation routines, was left unchanged. The Stream Characteristics Table had to be modified only slightly to include the new device. The Run Time Information Table for the Ramtek is simply a copy of the one provided for the Chromatics driver. The same is true for O4LIB1 with the exception of the O4EXEC routine. Modifications to O4EXEC were very slight. Inclusion of the new O4EXEC routine did necessitate changes to the VGM software in the INISTR and SELSTR (SELect STRing) modules.

The bulk of the work in writing the new driver will be in developing a new O4LIB2. Creation of a new Device Characteristics Table is also important. The latter requires an item by item reviewing of the table variables and providing the appropriate values as they pertain to the RM-9400. Details of what entries should be made are in the BNR "Skeleton Driver Installation Manual". In this study very little of the RM-9400's capabilities were exercised and the importance of the Device Characteristics Table was secondary to getting a rudimentary system in operation. As the device driver develops into a more refined form and incorporates more of the RM-9400's capabilities, the Device

Characteristics Table should be reviewed and updated continuously.

B. RAMTEK RM-9400 INSTRUCTION FORMAT

Before writing any of the routines, a knowledge of the format of an RM-9400 instruction is required. The RM-9400 instruction is a variable length buffer of 16 bit words. The first 8 bits of the first word contain the code for the function the device is to execute. For exa ple, an operation code of 06H (Ramtek mnemonic INIT) initializes the device; a code of 35H (Write Vector, Unlinked — mnemonic WVU) is a line drawing instruction. The operation code determines how much more of the instruction buffer the RM-9400 needs. For a very simple instruction like INIT, once the operation code is read, the rest of the buffer is ignored. For something like a line drawing, however, the device will require a number of extra words of information before it can execute the instruction. The extra words will, as a minimum, have to indicate the end points of the line, and may include the foreground and background colors as well as the line style and thickness. Depending on the operation, some of the additional instruction words are essential and some are optional. In most cases, if a piece of data is required and is not specified, a default value will be used. The architecture of the RM-9400 is such that it is possible for the user to specify the default parameters values.

The presence of additional words of information is indicated to the Ramtek by flag bits. The last 8 bits of the first word are the primary set of flags. Only three of these are involved in extending the size of the instruction. The five high order bits of the low byte modify the execution of the operation code as indicated in the following table.

bits 6 & 7	code for the mode of addressing refresh memory
bit 5	selects additive mode of text output
bit 4	reverses foreground and background colors
bit 3	sets device to process bytes in a word in reverse order

It is the three lowest bits that indicate how many additional words of data will be in the instruction. Bits 2 and 1 each indicate whether a particular "operand flag word" will be in the instruction buffer.

If bit 1 in set, "operand flag word #1" (OF1) will immediately follow the word containing the operation code and the primary flags. OF1 is another set of flags each of which corresponds to additional pieces of information in the instruction. The setting of one of the 16 bits in OF1 means that one or more words of data will follow. The order of the additional data will correspond to the bits of OF1, from lowest to highest.

A sample bit pattern might be

0000 0001 0000 0010 (0102H)

This would indicate that two additional pieces of data (one for each "1" bit) are in the instruction buffer following OF1. In this case the two bits that are set mean that "foreground" data and "line dimension" data are present (bits 1 and 8 respectively). The foreground information is simply an integer index to a color table in Ramtek refresh memory and only requires a single word of storage. The line dimensioning information is more complex consisting of a pattern code and a repeat code (interpretation of these codes is not important to the current discussion) and requires two words. These two words will follow the foreground word in the buffer. The code that the RM-9400 would execute is shown symbolically in Figure 5.

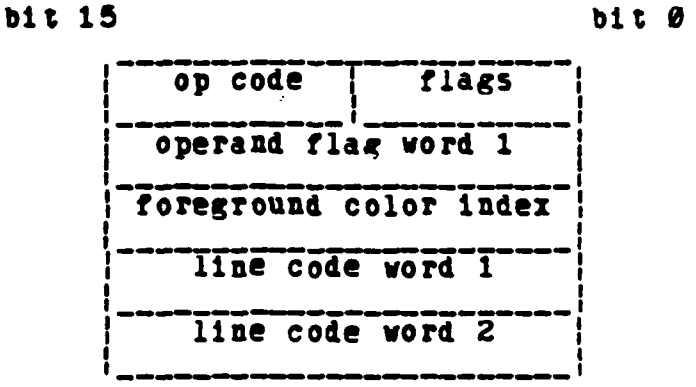


Figure 5. Instruction Buffer with OF1 and Associated Parameters

Bit 2 of the primary flags indicates the presence or absence of "operand flag word 2" (OF2). If flag bit 2 is set

then OF2 will immediately follow OF1. If flag bit 2 is set and flag bit 1 is not, then the word immediately following the operation code word is OF2. OF2 performs exactly the same task as OF1 except that only its six least significant bits are used.

Flag bit 0 in the operation code word indicates that text or numeric data will follow OF1, OF2 and their associated parameters if present. The first word of this data will always be an integer indicating how many additional bytes of data will come after it.

To illustrate, suppose the user wants to draw a line from the pixel at screen coordinates 100,100 to the one at 500,500. He further want its color to be that of #4 in the color table in refresh memory and his line dimension code is given by the words 0101H and 0003H. The instruction buffer would be that shown in figure 6.

To a rough approximation the operation codes for RM-9400 instructions correspond to VGM primitive and device control functions. Similarly, the parameters referenced by the operand flag words correspond to attribute values. While this is not always the case it provides a good rule of thumb for the initial development of a device driver. Those VGM calls executing a primitive or control function will cause an operation code to be placed in the instruction buffer. Those setting an attribute will cause the setting of flags and loading of appropriate parameter values.

C. SETTING ATTRIBUTES

1. Storage Requirements

When an attribute is set it does not necessarily have to take effect immediately. Therefore memory locations must be available to store the values of attribute settings until an appropriate operation code using them is loaded

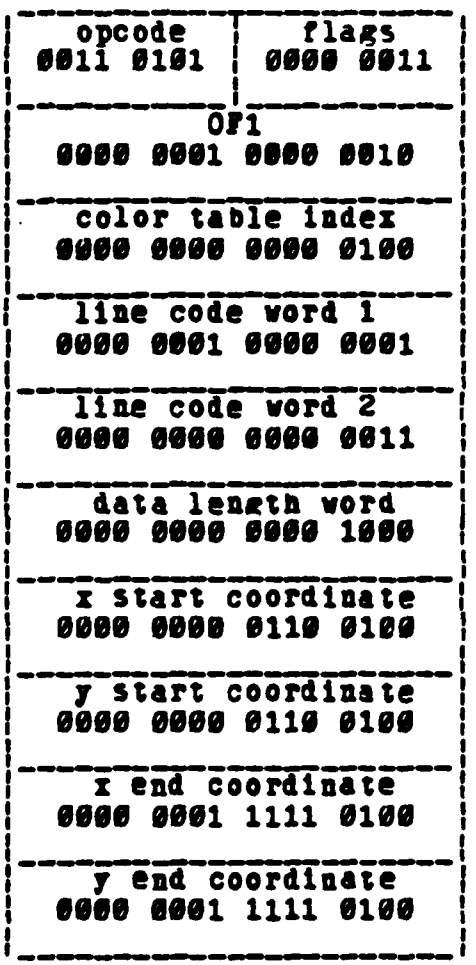


Figure 6. Instruction buffer for line with specified color and style

into the instruction buffer. The RM-9400 has the capacity for setting 22 different parameters, each one corresponding to a bit in one of the operand flag words (16 in OF1 and 6 in OF2). Not all of the parameters that are entered into the Ramtek instruction are limited to a single word. Although some parameters do require only one word there are others that need 2 or 4, and in one case 12, words of storage. The total storage required for the 22 parameter settings is 47 words. A global, integer array of 47 elements named PARAM is set up as part of a BLOCK DATA program to store these values as they are set. Two additional "read only" arrays are also part of that BLOCK DATA subroutine. These aid in referencing the PARAM array. Integer array PRMPTR of length 22 contains indices to the starting location in PARAM where a particular parameter's values are saved. Each entry in PRMPTR corresponds to a flag in one of the operand flag words. Elements 1 through 16 of PRMPTR contain pointers to the data associated with bits 0 through 15 of OF1. Elements 17 through 22 do the same for bits 0 through 5 of OF2. The second array, PRMSIZ, is a parallel array to PRMPTR. Each of its integer values is the size, in words, of the particular parameter pointed to by the corresponding element in PRMPTR. The values of PRMPTR and PRMSIZ are fixed at compile time and remain unchanged throughout the program. PARAM is initially set to all zero entries and gets updated as the various attribute values are assigned.

There are also two logical arrays and three logical variables involved in keeping track of attribute settings. The elements of the logical arrays correspond to the bits of OF1 and OF2. They are named OF1ARR and OF2ARR. If a bit in either OF1 or OF2 is to be set by a particular subroutine call then the element in OF1ARR or OF2ARR that corresponds will have a FORTRAN value of .TRUE.. The logical variables correspond to the lowest three bits of the primary flag set of the Ramtek instruction operation code word. For bit 2 there is OF2FL, for bit 1 there is OF1FL and for bit 0 there is DFFL. All of the logical variables, including the arrays are initialized to .FALSE..

Finally, the three words which actually control the instruction buffer size must be kept: OF1, OF2, and DLW (data length word). These are declared as integer variables in the BLOCK DATA subroutine and initialized to zero.

2. Value Storage Operations

When a subroutine that sets an attribute is called, several operations must take place, not necessarily in any particular order. One of the required events is that the attribute values must be entered into the proper location in PARAM. This is done in a special subroutine called O4LOAD. Its essential part is a DO loop which fills the array starting at the location indicated by an element of PRMPTR and filling the number of words indicated by the corresponding PRMSIZ value. Along with setting the parameter

values a flag marking the fact that they are to be used must also be set in either OF1ARR or OF2ARR. Also, the flag indicating either OF1 or OF2, must be set, so either OF1FL or OF2FL becomes .TRUE..

The code for any subroutine that sets an attribute can be written according to the following algorithm:

```
input: parameter values
begin attribute setting subroutine
    set OF1FL or OF2FL
    if flag in OF1ARR or OF2ARR not set then
        set OF1ARR or OF2ARR flag
        set flag in OF1 or OF2 by adding proper power of
            two
    end if
    store parameter values in PARAM
end
```

Two subroutines have been written that carry out this process. They are O4COLR and O4BCOL. The source code for them is found under RSX on the PDP-11/50 in directory DP3:[301,1]. The subroutine O4LOAD is also found in this directory.

3. Filling the Instruction Buffer

The critical subroutine in module O4LIB2 is COPCOD. This is the one that controls the filling of the instruction. It is invoked by the primitive and control subroutines in O4LIB2. The routines that call COPCOD pass as a parameter an index to an array containing the Ramtek operation codes. These operation codes are set into the array OPCODE at compile time. They are designed to be the Ramtek equivalent of the intended VGM function. COPCOD gets

the actual code from the array OPCODE. The operation code and the primary flags are then combined into the first word of the instruction buffer. COPCOD also causes the operand flagwords (OF1 and OF2), their parameter values and the data length word to be loaded if any of them are required. The text or numeric data is loaded by the primitive or control routine itself.

The following algorithm shows the operation of COPCOD. The code can be found in directory DP3:[301,1]:

```
input: OPCODE index
begin COPCOD
    get actual operation code from array OPCODE
    if OF1FL = true then set flag bit 1
    if OF2FL = true then set flag bit 2
    shift operation code to upper byte of first
        instruction word (multiply by 256)
    add flagword to shifted operation code
    load operation code word into buffer
    load OF1 and/or OF2 as indicated by flags
    load parameters in proper order from PARAM
    load data length word if necessary
end
```

As with attribute setting, all subroutines which are primitive functions can be patterned after a single algorithm. Control functions can follow this same pattern though they typically will not require data values to be placed in the instruction buffer. The algorithm is:

```
input data values
begin function execution
    set DFFL (data flag) if appropriate
    set bit 8 of flags
    set data length word (DLW) to number of bytes of data
    call COPCOD -- pass OPCODE index
    load data values
    execute instruction
end
```

A number of routines have been written following this algorithm. They include:

O4DOT	place a point at specified coordinates
O4MOVE	change current cursor position
O4DRAW	draw a line between two specified points
O4SSTR	output a text string
O4RSET	erase the screen.

All can be found in directory DP3:[301,1]. It should be noted that the routine O4STR, which builds an instruction for text output, inserts an extra step before loading the data. O4STR receives its text data in an integer array with one alphanumeric character per element. The Ramtek requires textual data output to be formatted to one character per byte. Therefore O4STR must make this conversion before loading the data into the buffer.

4. Testing

As the routines were developed they were tested for operability. The testing was at a very low level simply checking that the instruction buffer was being properly constructed and transmitted to the RM-9400. All of the routines developed in this portion of the study have been successfully checked in this manner. The test programs are modularized, each being called by a master routine called DUMTST. The individual test routines are named LINTST (LINe drawing TeST), PTTST (PoinT placement TeST), TXTST (TeXt

output TeST) and COLTST (COLor selection TeST). Their source code can be found in directory DP3:[301,1].

All of the routines developed were originally part of the O1LIB2 module of the Chromatics driver. A copy of the original software is available in directory DP3:[5,3]. After modifying the names to conform with the selection of #4 as the identifier for the new device driver each subroutine was removed from the module and treated as a separate entity. This was done for ease of editing and troubleshooting. To support this process command files were created to facilitate repetitive compilation and task building operations. The files COMP.CMD and TCOMP.CMD in directory DP3:[301,1] cause the compilation of all the driver software and all the test software, respectively. File DUM.CMD builds the test task DUMTST.TSK, by linking the modified subroutines, the BLOCK DATA programs and the test routines.

Among the test routines some short pieces of code have been included to accomodate testing. The important ones are RMINIT, which inserts the color table into the Ramtek refresh memory, and BIGTXT, which causes text output to be printed in a larger than normal format for viewing ease.

APPENDIX C. PROGRAMMING WITH VGM

Graphics programming using VGM is actually a highly specialized use of FORTRAN. Because of this, all the rules of PDP-11 FORTRAN apply as well as those of the graphics system. When using VGM the programmer generates a main program which references a library of graphics subroutines.

A. SETUP

VGM is initialized by three essential statements. First

```
CALL INIT
(INITialize)
```

starts the VGM session. It sets the variables required by both the operating system and VGM itself. The routine ensures that each time VGM is invoked it is in the same starting state. Immediately following this,

```
CALL INISTR(n)
(INItialize STReam)
```

activates the device dependent driver software for stream 'n'. It sets device parameters and uses the operating system process handling capabilities to allow application program access to the particular device or devices on the stream. Lastly,

```
CALL SELSTR (n1)
(SELect STReam)
```

directs all graphics output to stream 'n1'. If another CALL SELSTR (n2) instruction is encountered before a CALL

DELSTR(n1) instruction, the graphics output will go to both streams n1 and n2. It is mandatory that each stream be prepared by a CALL INISTR(n) before it is selected by a CALL SELSTR (n).

B. ENVIRONMENT SPECIFICATION

1. The Coordinate System

In VGM, the user defines his graphical world in arbitrarily selected "world coordinates". These coordinates are the medium through which he communicates positional data to the system. VGM processes those world coordinates through a series of viewing transformations and eventually derives "normalized device coordinates" (NDC). The NDC's are real values ranging from 0.0 to 1.0 and are mapped onto the physical device selected for viewing. A picture in NDC is independent of any particular graphics device.

For VGM to properly execute the string of required coordinate transformations, the Normalized Device Coordinate System must be specified before World Coordinates. With

CALL NDCSPC (nstrm, width, height)
(Normalized Device Coordinate SPeCification)

a rectangular portion of the view surfaces of terminals on stream 'nstrm' is defined. 'Width' and 'height' are real numbers ranging from 0.0 to 1.0. They indicate the relative part of that dimension of the view surface that is to be used. One or the other must be 1.0. Therefore, either the full screen width or the full screen height will be used.

The remaining dimension will be proportionately adjusted. A statement such as

CALL NDCSPC (1,1.0,.75)

will set up the devices on stream #1 so that a viewing area using the full width of the screen is made available. The height will be 3/4 as large as the width if that much is available. If a dimension specification is too large, the maximum available is used. The NDCSPC command normally is used only once for each stream.

2. The View Surface

After setting the total viewing area available, "viewports" are assigned to the streams by

CALL VIEW (nstrm, xmin, ymin, xmax, ymax)

A viewport is a portion of the available surface (NDC space) that will be used, up to, but not exceeding, the total declared surface. 'Xmin', 'ymin', 'xmax', and 'ymax' are NDC values and must be within the bounds specified in the NDCSPC call. A viewport declaration stays in effect until a new one is declared. The viewport may be changed as often as the user desires in the main program.

The "window" is the counterpart of the viewport in the world coordinate system and is set by

CALL WINDOW (nstrm, xmin, ymin, xmax, ymax)

The parameter values except for 'nstrm' are in World Coordinates and are arbitrarily selected by the user to meet his requirements for clipping and image transformations.

This is the part of World Coordinates that the picture is created in. The window is the work area of the programmer. For display, the clipped and transformed images in the window are mapped to the NDC space defined by the viewport.

3. Background Color

On color capable devices the last environment setting operation is to define the background with

```
CALL BCKCOL(ival)
(BaCKground COLor)
```

to set the attribute and

```
CALL ERASE
```

to bring it up on the screen.

The current version of VGM allows selection of one of only eight colors available. Selection is done by specifying an integer value between 0 and 7 for 'ival'. The default color table contains black, blue, green, cyan, red, magenta, yellow, and white, in that order.

C. CREATING A PICTURE

With the devices initialized, and the environment set, the next step is to open a segment. To do this the type of the intended segment must first be declared by

```
CALL SEGTYP (itype)
(SEGment TYPe).
```

An 'itype' value of 1 indicates that all subsequently created segments will be non-retained. A value of 2 means that they will be retained.

After the type has been established, the segment is opened with

CALL CRESEG (nsegmt)
(CREate SEGment).

'Nsegmt' is an integer value that uniquely identifies the particular segment. Once the segment is open, the user creates his image by invoking primitives and assigning attributes. Any primitive attribute values declared before closing the segment are static. Declaration of segment attributes is not allowed while a segment is open. As each primitive is executed its contribution to the total image will be displayed. When the particular image is completed, the

CALL CLOSEG
(CLOse SEGment)

instruction is issued. It is not necessary to specifically identify the segment being closed, since only one is allowed to be open at any given time.

After closing a segment, a number of options are open to the user. He may terminate the session by normal FORTRAN procedures or he may continue on and manipulate devices, streams and segments. He may alter dynamic attributes and create additional segments subject to the limitations of VGM and FORTRAN.

D. EXECUTING THE PROGRAM

After creating the source code for a VGM program it should be independently compiled into an object file.

Incorporating the user file and the VGM library into an executable task requires the following command sequence to be issued to the RSX Task Builder (for purposes of the example, MAIN is selected to be the name of the user's program):

```
TKB>  MAIN/CP/FP,VGM/-SP=VGMLI/MP
      ASG = SY: 1
      ASG = SY: 2
      ASG = SY: 4
      ASG = TI: 5
      ACTFIL = 3
      MAXBUF = 80
      FMTBUF = 80
      //
```

Before executing the task (now stored in file MAIN.TSK) it is necessary to INSTALL the device drivers. For each stream that will be used by MAIN.TSK. The MCR command to RSX is

```
>INS OnDRIV
```

where 'n' is the integer identifier for the particular driver. The command

```
>RUN MAIN
```

will execute the user's graphics program.

BIBLIOGRAPHY

Michener, J.C., and Foley, J.D., "Some Major Issues in the Design of the CORE Graphics System", Computing Surveys, 10, 4, pp. 389 - 443, (Dec 1978).

Michener, J.C., and VanDam, A., "A Functional Overview of the CORE System", Computing Surveys, 10, 4, pp. 381 - 387, (Dec 1978).

Newman, W., and VanDam, A., "Recent Efforts Toward Graphics Standardization", Computing Surveys, 10, 4, pp. 365 - 380, (Dec 1978).

Newman, W.M., and Sproull, R.F., Principle of Interactive Computer Graphics, McGraw-Hill, New York, New York, 1979.

"Status Report of the Graphics Standards Planning Committee of ACM/SIGGRAPH", Computer Graphics, 11, 3 (Fall 1977).

"Status Report of the GSPC", Computer Graphics, 13, 3 (August 1979).

Bell Northern Research, Virtual Graphics Machine: User's Reference Manual; Installation Guide; Skeleton Driver Installation Guide, Ottawa, Canada, April 1981.

Digital Equipment Corporation, PDP-11 FORTRAN Language Reference Manual, Maynard, Mass., 1975.

Ramtek Corporation, RM-9400 Series Graphics Display System, Software Reference Manual, Santa Clara, Ca. 1979.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Professor George A. Rahe, Code 52Ra Department of Computer Science Naval Posgraduate School Monterey, California 93940	1
5.	Lt. Patrick M. Comi, USN 3938 Via de la Bandola San Ysidro, California 92072	1

END

FILMED

1-83

DTIC